GVTDOC
D 211.
9:
3531

NAVAL SHIP RESEARCH AND DEVELOPMENT CENTER
Bethesda, Maryland 20034

DESIGN TRADE-OFFS
FOR A
SOFTWARE ASSOCIATIVE MEMORY

LIBRARY
AUG 31 1973
U.S. NAVAL ACADEMY

by

S. Berkowitz, Ph.D.

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

COMPUTATION AND MATHEMATICS DEPARTMENT
RESEARCH AND DEVELOPMENT REPORT

20070119024

May 1973

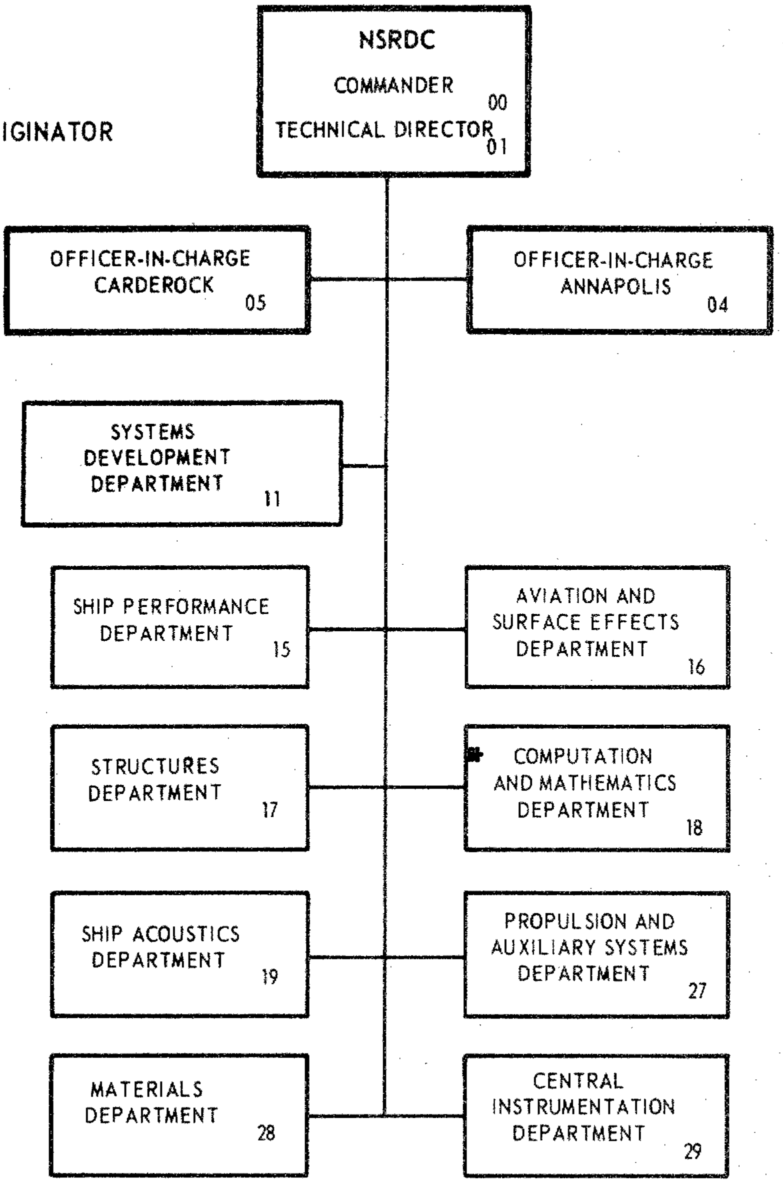
DESIGN TRADE-OFFS FOR A SOFTWARE ASSOCIATIVE MEMORY

The Naval Ship Research and Development Center is a U. S. Navy center for laboratory effort directed at achieving improved sea and air vehicles. It was formed in March 1967 by merging the David Taylor Model Basin at Carderock, Maryland with the Marine Engineering Laboratory at Annapolis, Maryland.

Naval Ship Research and Development Center
Bethesda, Md. 20034

MAJOR NSRDC ORGANIZATIONAL COMPONENTS

*REPORT ORIGINATOR


NSRDC
COMMANDER 00
TECHNICAL DIRECTOR 01
OFFICER-IN-CHARGE CARDEROCK 05
OFFICER-IN-CHARGE ANNAPOLIS 04
SYSTEMS DEVELOPMENT DEPARTMENT 11
SHIP PERFORMANCE DEPARTMENT 15
AVIATION AND SURFACE EFFECTS DEPARTMENT 16
STRUCTURES DEPARTMENT 17
* COMPUTATION AND MATHEMATICS DEPARTMENT 18
SHIP ACOUSTICS DEPARTMENT 19
PROPULSION AND AUXILIARY SYSTEMS DEPARTMENT 27
MATERIALS DEPARTMENT 28
CENTRAL INSTRUMENTATION DEPARTMENT 29

DEPARTMENT OF THE NAVY
NAVAL SHIP RESEARCH AND DEVELOPMENT CENTER
BETHESDA, MD. 20034

DESIGN TRADE-OFFS
FOR A
SOFTWARE ASSOCIATIVE MEMORY

by

S. Berkowitz, Ph.D.

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

May 1973

Report 3531

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This report describes an associative (content-addressable) computer memory simulation, called GIRS (Graph Information Retrieval System), designed to handle the dynamic insertion, retrieval, and deletion of arbitrary symbolic or numeric data structures. The main purpose of the study is to demonstrate fundamental trade-offs between time, space, complexity, and flexibility in the field-level operation of any associative memory simulation. Specifically, the paper concludes that

- a reduction of retrieval time is possible at the cost of a complex linkage scheme and slow insertion;
- the design of a random node generator can be optimized to match the scrambling transformation and reduce retrieval time;
- a dynamic reorganization of pages and the use of inference mechanisms can reduce the number of page fetches and handle complex queries with minimal storage.

ADMINISTRATIVE INFORMATION

This report was written in the Computer Sciences Division of the Computation and Mathematics Department under the sponsorship of NAVSHIPS 0311, Task Area SR0140301, Task, 16565, Element 61151N, Job Order 1-1834-001.

INTRODUCTION

This paper describes an associative (content-addressable) memory simulation, called GIRS (Graph Information Retrieval System), designed to handle the dynamic insertion, retrieval, and deletion of arbitrary symbolic or numeric data structures. The main purpose of the paper is to demonstrate some fundamental trade-offs between time, space, complexity, and flexibility in the field-level operation of an associative memory simulation. Specifically the paper investigates

- reduction of retrieval time versus the cost of a complex linkage scheme and slow insertion;
- optimal design of a random node generator to match the scrambling transformation and reduce retrieval time;
- dynamic reorganization of pages and the use of inference mechanisms to reduce the number of page fetches and to handle complex queries with minimal storage.

In an ideal associative memory, every address (called a node address, or simply, a node) has a potential labelled association with every other node in the memory. When an association is realized say by placing the label and the node couple in a cell--one says that an association, or link, has been formed between a source node and a sink node. The abstract representation of such a link is called a node-link-node triple (Figure 1).

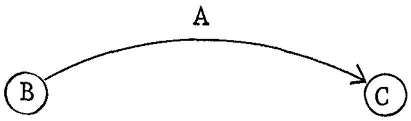


Figure 1 - Node-Link-Node Triple

Other terminology treats the link in the node-link-node triple as a function in which the argument is the source node and the value is the sink node. Indeed, the function may be multivalued, as shown in Figure 2.

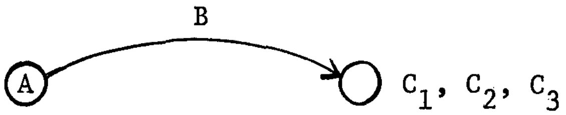


Figure 2 - Multivalued Function

The values are then said to be on a *value list*. The values may be nodes (i.e., cell addresses), numeric (e.g., integer) data, or Hollerith data.

In a computer having random access sequential memory, every memory cell is directly accessible to the system software through a numerical address. The memory is *sequential* due to the complete ordering of addresses, and *randomly accessible* (rather than serially accessible), since any address can be specified directly without regard to its ordered antecedents. Other than for the ordering of addresses, there is no hardware linkage between cells.

Due to its rigid but easily accessible structure, a sequential memory is especially well suited for numeric processing of fixed data arrays. Most modern computers have sequential memories. On the other hand, for non-numeric data processing or processing of dynamically changing data structures, one may require an associative memory. In order to simulate an associative memory in a sequentially addressed computer by means of software, it is most economical to employ a *content-addressable* memory scheme, the nature of which will be reviewed in the next section.

Diagrams composed of combinations of node-link-node triples represent *graph structures*. GIRS is an associative (content-addressable) memory scheme designed to accommodate the insertion, retrieval, and deletion of information contained in arbitrary graph structures. These operations are dynamic in the sense that space for a particular association or list involved in the operation need be reserved only at execution time and not by declaration beforehand.

Morris[1] has written an excellent review on the subject of content-addressable memories. That review is summarized in the next section. Associative languages have been discussed by Feldman and Rovner[2] and Berkowitz.[3,4] In addition, a language with inferential capability is discussed in Ash and Sibley.[5]

BACKGROUND ON CONTENT-ADDRESSABLE MEMORIES

A review by Morris on the subject of content-addressable memories is summarized here to focus attention on the critical design elements of a content-addressable memory.

COMPUTATION OF HASHED ADDRESSES

With each block of data A_1 to be stored, we associate a key, A_2. The key may be extracted in part or in its entirety from the data. If the block of data extends over several memory cells, some internal linking for the block must be employed, but, for the purposes of the discussion, it suffices to consider (A_1, A_2) as occupying one cell.

A hashed or scrambled address, which indicates the location of the data A_1, is the value of a transformation T of the key A_2. The transformation T is single-valued but, in general, many-one, so that in order to avoid many collisions of hashed addresses, it is wise to select a T which provides a uniform spread of hashed addresses throughout memory. The resolution of collisions is discussed later. One way to choose a transformation is to

[1] References are listed on Page 50 in the order of their occurrence.

select some bits from the middle of $(A_2)^2$ or from the product or sum (modulo the memory size) of terms of a subset of A_2--hence the description hashed or scrambled address. The resultant apparently uniform random spread of addresses is called a scatter storage. Since, excepting a collision, the key A_2 is stored at the address $T(A_2)$ or is implicit, in part, in the address itself, the hashed address is a function of its contents--hence the description content-addressable memory.

RESOLUTION OF COLLISIONS

When two keys A_2 and B_2 collide--i.e., have the same hashed address, $T(A_2) = T(B_2)$--the data block B_2 must be reassigned to another address. The resultant list of collisions is called a conflict list.

There are two classes of techniques for constructing conflict lists: increment probing, and direct chaining.

In the increment probing technique, one searches or probes the memory for an empty cell that can be added to the conflict list by incrememting the last conflict address x_i as follows:

$$x_{i+1} = x_i + f_i \quad (\text{Mod } M)$$

where M is the memory size and f_i may be

- a fixed constant (If $f_i = 1$, the search is called linear probing);
- a function of i;
- a pseudo-random number (based either on a universal initiator--in which case the i^{th} increments of all conflict lists are the same--or on x_0 or some x_i as the initiator); the pseudo-random generator must be capable of generating M numbers without cycling.

In the direct chaining technique, an internal link A_3 is added to the keyed data (A_1, A_2) to indicate the location of the next cell on the conflict list. The conflict list is then chained, so to speak. The chaining may be one-way, two-way, or circular, depending on whether the links point down, or up and down, or have no termination (i.e., last link points to top cell), respectively. Variations of direct chaining depend on the use of overflow (generally secondary storage), as follows:

(i) Overflow may be used as a direct extension of scatter storage.

(ii) Scatter storage contains only heads of conflict lists, while overflow holds the list bodies.

(iii) Scatter storage holds only pointers to the heads of conflict lists in overflow: called scatter index table.

(iv) Scatter storage may be used as in (ii) or (iii) but with the hash address computed for a much larger scatter storage than is actually being used: called virtual scatter table or virtual scatter index table, respectively. This technique drastically reduces the number of conflicts, on the average, since the probability of collision is based on the larger scatter storage. On the other hand, a somewhat longer key must be used to account for the larger virtual storage.

Increment Probing Evaluation. The prime advantage of increment probing is the ease of programming. However, the disadvantages are several.

- Due to the intersection of conflict lists, the key must be entirely and explicitly stored with the data in order to resolve possible ambiguities of identification.
- Deletion of an entry requires either that (1) the gap in the conflict list be closed--at great computational expense--in order that a list termination not appear in the middle of a list, or that (2) a distinguished symbol or flag be placed in the empty cell to denote continuation, in which case space will be wasted and probe and deletion time somewhat increased.
- The average probe time for increment probing is somewhat greater than that for direct chaining.
- There is no convenient provision for overflow.

Direct Chaining Evaluation. The advantages of direct chaining are threefold:

- There is a savings in probe time over that for increment probing (especially in the case of virtual storage with its small conflict lists).
- There is a provision for overflow.
- In the last three of the overflow chaining techniques ((ii), (iii), and (iv)), an item once entered into a cell need never be moved to make room for a new insertion, since intersecting conflict lists do not occur. There is no need, in any of the chaining techniques, moreover, to close the gaps made when items are deleted, since one can chain around deleted entries. Deletion is simplified.

As might be expected, however, there are some complementary disadvantages, namely:

- More storage is required for links and also for the index table (where used); moreover, in all uses of direct chaining employing virtual storage, the key must be longer, as mentioned above.
- The programming is complex.
- Due to the linkage update necessary, insertion generally takes much longer than retrieval.

THE GIRS SCHEME

IN-CORE STORAGE DESIGN

GIRS is a scheme for the computer simulation of an associative, content-addressable, memory designed to store information structured as node-link-node triples. A direct chaining approach without paging is discussed first. Later, an extension to GIRS is discussed which involves the incorporation of paging procedures to accommodate programs requiring larger amounts of storage than is available in primary memory.

Available Space (free storage). Before any entry into GIRS is made, the sequential memory is organized into a circular two-way (up-down) list as shown in Figure 3. This arrangement is called available space (AS). The zeroed fields shown in Figure 3 will be discussed later. A distinguished register called REGASP (register of available space) holds the entry cell to AS. In general, by an initial rearrangement of AS under a permutation σ, one may be able to reduce the number of cell contents displacements--bumpings--caused by allocation of space for conflict list head cells in space already occupied by other type cells. For example, if hashed address generation is initially dense at low numbered addresses, it will be profitable to allow σ(A) = M-A for A<<M. For a sufficiently wide range in the data, the density of hashed addresses will be uniform on the average, and σ might as well be the identity.

σ(1):	(σ(2)	σ(N)	0	0)
⋮	⋮	⋮	⋮	⋮
σ(x):	(σ(x+1)	σ(x-1)	0	0
⋮	⋮	⋮	⋮	⋮
σ(N)	(σ(1)	σ(N-1)	0	0

Figure 3 - Initial Arrangement of Available Space

Work Space (bound storage). When an entry is made into the memory, one says that the entered cell has been removed from available space and placed in work space (WS). A cell in WS is structured in a four-field format as shown in Figure 4. As will be made clear, work space contains not only conflict lists, but also multivalue lists. The multivalue lists contain either sink nodes, numerical data, or Hollerith data.

x KEY/MVL MVLL/UV CLL F

Figure 4 - Work Space Cell Format

As an alternative means of realizing the complementary WS and AS, the GIRS system has a block of sequentially ordered space called sequential space (SS) in which multivalue list nodes or data may, optionally, be stored. Sequential space has the advantage of rapid list access and immediate overlay extension, and the corresponding disadvantage of necessarily predetermined fixed list lengths which may be established during execution but cannot be easily changed. Each block of SS is preceded by a header cell which contains the number of items in the block. Blocks are homogeneous as to cell type. In particular, any Hollerith insertion of more than ten characters constitutes a block of SS. When data is intended for SS, a pointer to the block in SS is left in the graph, as described below.

The WS fields in Figure 4 are defined as follows:

KEY/MVL Field (The key or multivalue list field) - Contains either

- A source node address (which also serves as a key) for all heads of univalue or multivalue lists, or
- A value in the body of a multivalue list; the value may be a sink node address, a number, or a pointer to an SS block.

MVLL/UV Field (The multivalue list link or univalue field) - Contains

- The addresses of the next cell and of the previous cell on the multivalue list. A two-way list is needed because of the potential length of multivalue lists. As will be shown later, the two-way list allows the efficient use of a list index operation; moreover, one can take advantage of the storage requirements for the CLL field, as will be described later.

- The value (sink node address, integer datum, or pointer to an SS block) on a univalue list.

CLL Field (The conflict list link field) - Contains the address of the cell next on the conflict list. When the current cell is the last conflict list cell, CLL contains the address of the head of the list. A circular list is feasible, since the conflict lists are generally very short (an average of two to three cells per list).

F Field (The flag field) - Contains eleven flags.

F0 - indicates whether or not the cell is head of a multivalue list;

F1 - indicates whether or not the cell is in WS;

F2 - indicates whether or not the cell is contained in the body of a multivalue list;

F3 - indicates whether or not the cell is a pointer to a multivalue list in SS;

F4 - indicates whether or not the address of the cell is defined to exist in the graph interpretation. This flag is used during clean-up of useless information--garbage-collection--so that cells may be returned to AS;

F5 - indicates whether or not the cell is head of a conflict list;

F6 - indicates whether or not the MVLL/UV field (in the case of a univalue list) or the KEY/MVL field (in the case of a multivalue list) contains a sink node;

F7 - indicates whether or not MVLL/UV or KEY/MVL or an SS block contains a Hollerith datum;

F8 - indicates whether or not MVLL/UV or KEY/MVL or an SS block contains a single-precision number;

F9 - indicates whether or not MVLL/UV or KEY/MVL or an SS block contains an integer, Actually, flags F6, F7,..., can be described by a binary number encompassing fewer bits than are required for flags. (i.e., flags F6, F7, F8, and F9 require a 2-bit number description). Note also that F0 = F2 = 0 implies a univalue insertion.

When a cell is situated at the head of a univalue or multivalue list, the CLL field is occupied by a link, as described above. However, when a cell is situated in the body of a multivalue list, the CLL field is normally empty. We can take advantage of this empty space for storing the up-pointers of the multivalue list. On the other hand, for a multivalue list of n cells (i.e., one which includes a head cell and n-1 value cells), there would be only n-1 up-pointers since the head cell CLL field contains a conflict list link. Therefore, one must tailor the up-down pointer structure to suit the operations proposed for the system, as detailed in the following discussion.

Since we must allow a conflict list head to replace, or bump, any cell of a multivalue list, the up (down) pointer of the succeeding (preceding) cell must be modified to point to the new body cell acquired from AS. If the body cell bumped happened to be the last cell of the list, we might require that the MVLL/UV field contain the address of the first value cell (i.e., the second cell on the list). Then, however, the CLL field of the first value cell would have to contain the head cell address in order to account for a possible bumping of the first value cell. This arrangement would be suitable if it were not for the fact that a basic motivation for having "up" pointers is that the bottom of the list be accessible from the head cell. We resolve the dilemma as follows: the CLL field of the first value cell contains the address of the bottom cell of the multivalue list; the MVLL/UV field of the last value cell contains the address of the head cell. Although this arrangement is sufficient and it avoids the difficulties just outlined, it nevertheless results in a structural asymmetry which makes detecting the ends of the up and down list scans difficult. Whereas the down-scan ends when the next address is that of the head cell (see flag F0), the up-scan must make a double access, one down to the bottom cell and the next up to the head cell. Unless we can find another means of handling conflict list links, this asymmetry is a necessary price for the flexibility of up-down lists.

Example of a GIRS Structure

In order to clarify the definitions and discussion just presented, we offer an example of a GIRS structure. Assume that the graph shown in Figure 5 is to be inserted into GIRS memory. The hashing transformation used by GIRS--to be discussed further in the next section--computes the address of an association by adding the source node and link addresses modulo the memory size. Since the transformation is invertible, only the source node address need actually be stored as the key.

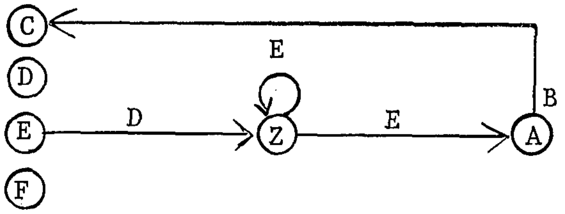


Figure 5 - Example Graph Structure

For an initial AS of 15 cells, and for the following address assignments,

A=6	B=2	C=4	D=3	E=5	F=1	Z=12

the storage configuration for the example graph of Figure 5 is shown in Figure 6 on the following page. For simplicity, we assume no (i.e., identity) permutation of the initial up-down list which forms AS. In the comments column of Figure 6, CL refers to the conflict list, and MVL to the multi-value list; the triple A, B, C, for example, refers to the insertion of the source node A linked by B to the sink node C. The order of insertions was: (A, B, C) (A, B, D) (A, B, E) (A, B, F) (Z, E, Z) (Z, E, A) (E, D, Z). Note that (A, B, C) was bumped by (Z, E, Z) and (Z, E, A) was bumped by (E, D, Z).

Memory	Cell Fields				Comment
ADDRESS	KEY/MVL	MVLL/UV	CLL	F	
1	6	6	8	1100100	body CL, head MVL
2	12	7	2	1100110	head CL, head MVL
3	3	4	6	0110101	A,B,D, body MVL
4	5	5	3	0110101	A,B,E, body MVL
5	1	1	4	0110101	A,B,F, last body MVL
6	4	3	5	0110101	A,B,C, first body MVL
7	12	9	9	0110001	Z,E,Z, first body MVL
8	5	12	1	0100011	E,D,Z, head CL, univalue
9	6	2	7	0110001	Z,E,A, last body MVL
10	11	15	0	0000000	AS
11	12	10	0	0000000	AS
12	13	11	0	0000100	AS
13	14	12	0	0000000	AS
14	15	13	0	0000000	AS
15	10	14	0	0000000	AS

Figure 6 - Storage for Example Graph

Another way to envisage the associations between the storage cells is by considering the 4-field cell pictured in Figure 7 in conjunction with the cell pointer graph of Figure 8 on the following page.

ADDRESS	KEY/MVL
MVLL/UV	CLL

Figure 7 - Cell Structure

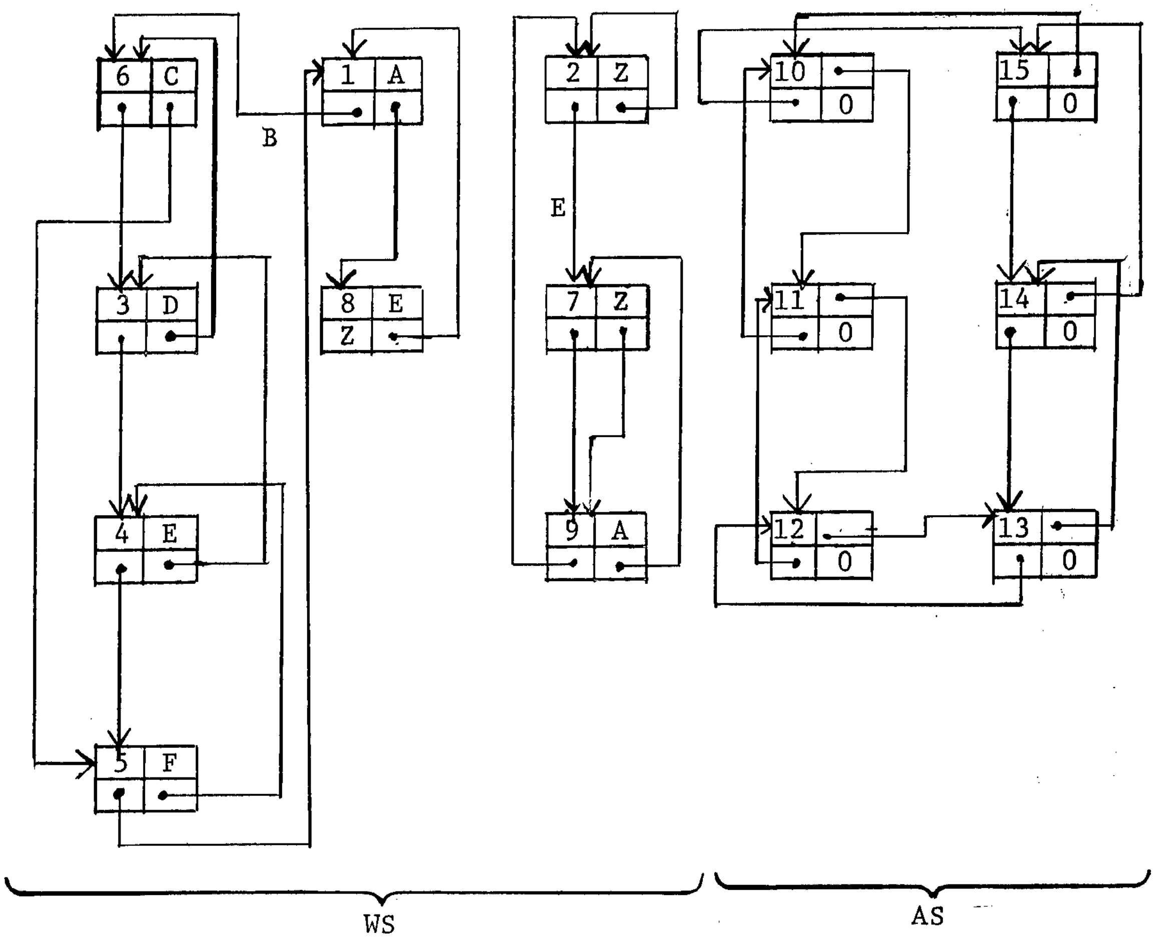


Figure 8 - Cell Pointer Graph

Before leaving the GIRS in-core storage design, we comment on the use and efficiency of the complex linkage just described. The multivalue list linkage provides a means of implementing list indexing in a high-level language[4]. In particular, the up-pointers give one the ability to implement LIFO (last-in first-out) pushdown stacks of dynamically modifiable length. At first glance, one might be concerned with the time required to access iteratively each member of a list--either up or down. Indeed, for a list of length k, one would make $\frac{k(k+1)}{2}$ accesses. However, we can remove this difficulty at compile time for GIRL[4] by generating local storage variables which hold the last index and list location referenced (together with the direction of list search) thus reducing the iteration.

to exactly k accesses. (The process is more complicated than what has just been described, but we leave that discussion for a report on the GIRL preprocessor or compiler.) In effect, then, the list access is a means of effecting matrix storage with dynamically modifiable dimension. Such storage is logically equivalent to the usual matrix access offered by most computers with sequential random-access memory in the sense that only one access is required, in general, per word retrieval--but the access, of course, takes more time due to the software implementation. Moreover, additional time is used for testing whether to count up or down from the head cell or from the last location used. The preceding techniques for the automatic retention of indexed multivalue list locations creates the subtle danger of a reference to a non-existent cell in the event that multivalue list cells are added or erased. Since the automatic updating of all indices referring to a list on which an addition or erasure occurs is a hopelessly complicated or inefficient task--for example, it might be done by setting all saved indices back to the first value--we are left with the caveat: programmer beware.

Finally, an alternative to index/location retention is the use of a fixed block of storage in sequential space (SS), as previously mentioned, to hold multivalue lists. As far as retrieval operations are concerned, SS will appear as a virtual WS. Insertion operations must specify the initial location of the block in a larger, previously dimensioned array called SEQ, and also the number of cells belonging to the block (to be inserted in the first location of the block). The latter device enables one to specify list termination and to count up or down the stored list. Links, of course, are not needed since the storage is sequential, and the only flags of interest are those which stipulate the type of cell entry. Unless the machine is byte-oriented, flag storage would pose an intolerable waste of space, and we are forced to declare a uniform data type for each block. The advantage of SS, of course, is the speed with which one can access list items. However, SS storage is somewhat rigid since the block length must be fixed. On the other hand, the length declaration is done at execution time and can be modified by the program.

Method of Operation

Retrieval. The process of retrieving the value(s) associated with the source A by link B is as follows:

1. If the "defined" flag F4 of either A or B is not set, then the process stops and the undefined status is reported outside for halting, garbage-collecting, or redefinition.

2. Compute the address of the head of the conflict list by the transformation

$$x = T(A,B) = A + B \pmod{M}.$$

Since the transformation T is invertible, B can be retrieved if necessary from x and A; therefore, one need have stored only A as the key in the K/M field.

3. Search the conflict list (having checked flag F1 of x to see if the list exists) for the key A by means of the links in the CLL field. The conflict list is circular so that retrieval failure is signalled by the second access of the list head (flag F5). If the key A is found, flag F0 will have been set if there is a multivalue list.

(a) If F0 is indeed set and F3 is not set, the multivalue list is accessed as a sequence of objects in the KEY/MVL field by links in the MVLL/UV field. The type of value accessed is determined by the flags F6, F7, F8, F9, which indicate whether the object is a sink node, Hollerith datum, single-precision number, or integer, respectively.

(b) If both F0 and F3 are set, the multivalue list is in the sequential space SEQ beginning at the address pointer located in the MVLL/UV field.

In all cases, undefinedness of a sink node is reported outside.

Insertion. The process of inserting the values C1, C2, C3 (C2, C3 possible null) associated with the source node A by the Link B is as follows:

1. The cell containing the key A in the KEY/MVL field is found, if it exists, exactly as for retrieval.

2. (a) If A is found, add the values C1, C2, C3 to the multivalue list by: extracting cells from AS (if not empty) through the REGASP entry; restoring AS; placing C1, C2, C3 in the K/M fields of the new cells. Set the up-down links in the MVLL/UV field, and set flags F1, F2, and F6-F9 the latter four depending on the types of C1, C2, C3. (Depending on the nature of the garbage collection routine, one also may wish to check F4 of A, B, C1, C2, C3, before searching the conflict list, and then delete the entry if A or B are undefined, or refuse to insert, if any value is an undefined node.)

(b) If the conflict list exists but A is not found, a new multivalue list must be set up as in (a), except that the first cell of the multivalue list must contain: A in the KEY/MVL field; the address of the head of the conflict list in the CLL field; either an up-down link in MVLL/UV for C2, C3 null, or C1 in MVLL/UV for C2 or C3 not null; and a setting of F1. Also, the CLL field of the last cell of the conflict list must be updated to contain T(A,B).

3. If the conflict list does not exist, one must be created beginning in T(A,B). The details are the same as those in (b) except that F5 must be set in the head cell. In the event that a cell of some multivalue or univalue list (head or body) occupies the space required by the new conflict list head cell, the offending cell must be removed from T(A,B) and placed in a cell drawn from AS. The flags and key (or value) are copied from the old to the new cell and the links are modified to preserve the multivalue and/or conflict list--the latter in the event the offending cell was the head of a univalue or multivalue list.

4. If one requires that the multivalue list exist in SS, the desired address index of SEQ, say I, is placed in the MVLL/UV field of T(A,B) (or its conflict list descendant containing the key A, as the case may be). The values are then located sequentially in memory beginning at SEQ(I+1). The number of block locations must be stored in SEQ(I). Hollerith insertions are always made into a block of SS.

5. If AS has been depleted, one must resort to an overlay, to a garbage collection, or to a processing halt.

Deletion. The process of deleting the values associated with the source node A by the link B is as follows:

1. A retrieval of the cell--on the conflict list headed by T(A,B)--containing A in the K/M field is performed exactly as in the retrieval or insertion procedures.

2. The cells on the multivalue list are returned to AS with F2 turned off by linking them through the access cell in REGASP.

3. The conflict list is restored by chaining around the gap left by the deleted value list head.

4. If the value list had been in SEQ (F3 set) the list head is simply returned to AS as above and no further action need by taken as regards AS. The head cells of any SS blocks on the multivalue list must be set to zero.

Effects of Node Generation on the Scrambling Transformation

The character of a good scrambling transformation T(A,B) is such that the average length of conflict lists is minimum. If the node generator is random with uniform density p(ß), then the density f(s) of the sum mapping T(A,B) = A+B(mod M) can be shown to also be uniform by Equation (1).

$$f(s) = \int_{-\infty}^{\infty} p(\beta)\, p(s-\beta)\, d\beta \qquad (\text{mod } M) \qquad (1)$$

One might then reasonably conclude that a uniform scatter of the association triplets would provide a satisfactory minimization of conflict list length. However, there are some difficulties with such a conclusion.

1. The primary objection to the use of a uniform random node generator is that it is not specifically matched to the mapping scheme. That is, a judicious choice of the generation sequence might result in a lower average conflict list length. Consider the node sequence $y_1, y_2, \ldots, y_m$. Let the probability (density) that for some y and for arbitrary $(y_i\ y_j)$, $T(y_i, y_j)$ maps onto y, be $p(y) \equiv p$. If the number of value list heads in WS is K, then the probability $p_k(y)$ of creating a conflict list at y of length k (for K>>k) is approximately Poisson. (We only consider value list heads since cells in the list tails may be bumped by conflict list items. It suffices, therefore, to consider the tails as being stored outside of WS--in SS, for example.)

The average conflict list length in WS (including zero lists) is then

$$K \left\{ \sum_{y=1}^{M} \left[\sum_{k=0}^{K} k\, p_k(y) \right] / M \right\}^{-1} = M \tag{2}$$

where $K \leq m^2$ (m being the number of nodes generated). If $K > m^2$, then K must be replaced by m^2 since there are at most m^2 distinct (source node, link) pairs available. The average length given by Equation (1) is independent of the sequence of nodes generated, (and of the specific number of nodes although m must exceed $\sqrt{K}$). This means that, for a sufficiently large WS (i.e., large K, m) the average access time will be the same for any generating scheme. Moreover, it is clear that if all the addresses of AS are generated as nodes, then the distribution of possible conflict lists is such that there is a list of length M possible at every address. This fact implies that, for a sufficiently large number of nodes generated, the distribution of conflicts tends to be uniform irrespective of the generating scheme. Therefore, if any generating scheme has an advantage, it must be that one can avoid many conflicts only before most of the node addresses are generated. Uniform random node generators do not take advantage of this fact.

We speak in the preceding about averages although there are, no doubt, optimum generating schemes for any particular program and data structures. Indeed, for certain problems, we have established compact graphs without conflict lists at all. Moreover, it is possible to adaptively select the next best node to fit the current node distribution, albeit at considerable computational expense. In general, however, it is not feasible to custom design the generating scheme.

2. Another objection to the use of the uniform random node generator is that it generates multiple hits before producing a complete sequence. Indeed, if t is the probability of generating a node address that has already been generated, then the average number of accesses required until an unused node is generated is

$$\sum_{i=1}^{\infty} i\, t^{i}(1-t) = \frac{t}{1-t} \qquad (3)$$

thus, as more nodes are produced, it becomes more time-consuming to generate the next one. Moreover, extra storage must be reserved for a flag bit which indicates whether or not a node has yet been generated.

3. Finally, the multiplication operation employed by uniform random generators is expensive, and as we shall see, it might be desirable to have, a generation scheme based on an addition operation.

We will now discuss a possible scheme to meet the objections just raised, and show that a variant of the scheme is acceptable.

For a given AS of length M, one could select a prime $P < M$; generate the additive group of modulus P with the knowledge that any s, $1 \leq s \leq P$. is a generator of the group; and complete the generation sequentially for the addresses P+1 through M. Although this scheme both avoids the multiple hit problem and also is efficient, it does not necessarily meet the primary objection of matching the scrambling transformation. Indeed, the scatter may produce strong clustering, depending on the generator s.

Suppose we first generate P numbers in the generator s_1, for P a prime, as just described. Call these numbers $s_1, s_2, \ldots, s_p$, where $s_j = js_1 (\text{mod } P)$. Then let each s_j serve as a further generator of a sequence

$$\left\{ s_{jh} \mid s_{jh} \leq M,\ s_{jh} = s_j + h^2/2 + (P-1/2)h - P \right\} \qquad (4)$$

The sequence in Equation (2) is in fact generated recursively using only additions, as we shall see. As shown in Figure 9, the sequence thus far generated leaves certain residues. These can be generated in a straightforward manner, as shown in the complete algorithm flow chart in Figure 10.

$$
\left.\overbrace{\begin{array}{llll}
s_1 & s_1+P+1 & s_1+2P+3 & \cdots \\
s_2 & s_2+P+1 & s_2+2P+3 & \cdots \\
\vdots & \vdots & \vdots & \\
s_P & s_P+P+1 & s_P+2P+3 & \cdots
\end{array}}^{\text{increasing increment}}\right\} \text{initial generation}
$$

$$
\left.\begin{array}{llll}
P+1 & 2P+2 & 3P+4 & \cdots \\
 & 2P+3 & 3P+5 & \cdots \\
 & & 3P+6 &
\end{array}\right\} \text{residues}
$$

Figure 9 - Node Generator Row Sequence

The generation scheme just described not only avoids multiple hits and multiplication, but also has a low average conflict list length, at least for small m, because of the increment increase of each column as shown in Figure 9. The algorithm is not as complex as Figure 10 makes it appear since the primary loop involves only a node update and node increment (dnode) update.

The average number of accesses for retrieval when the node-link pair is not in WS (called FIND^-), for retrieval when the node-link pair is in WS (called FIND+), and for insertion (called ADD) are now computed. The probability of producing a mapping $T(y_1,y_2)$ at a k-list address among all of AS/WS is $\sum_{j=1}^{M} p_k(y)$. But the probability of producing the same in WS alone is*

$$p(K) = \frac{k}{K} \sum_{y=1}^{M} p_k(y) \qquad (5)$$

* Equations (5)-(9) are generalizations of equations derived by Professor A. Newell of Carnegie-Mellon University, Pittsburgh, Pennsylvania.

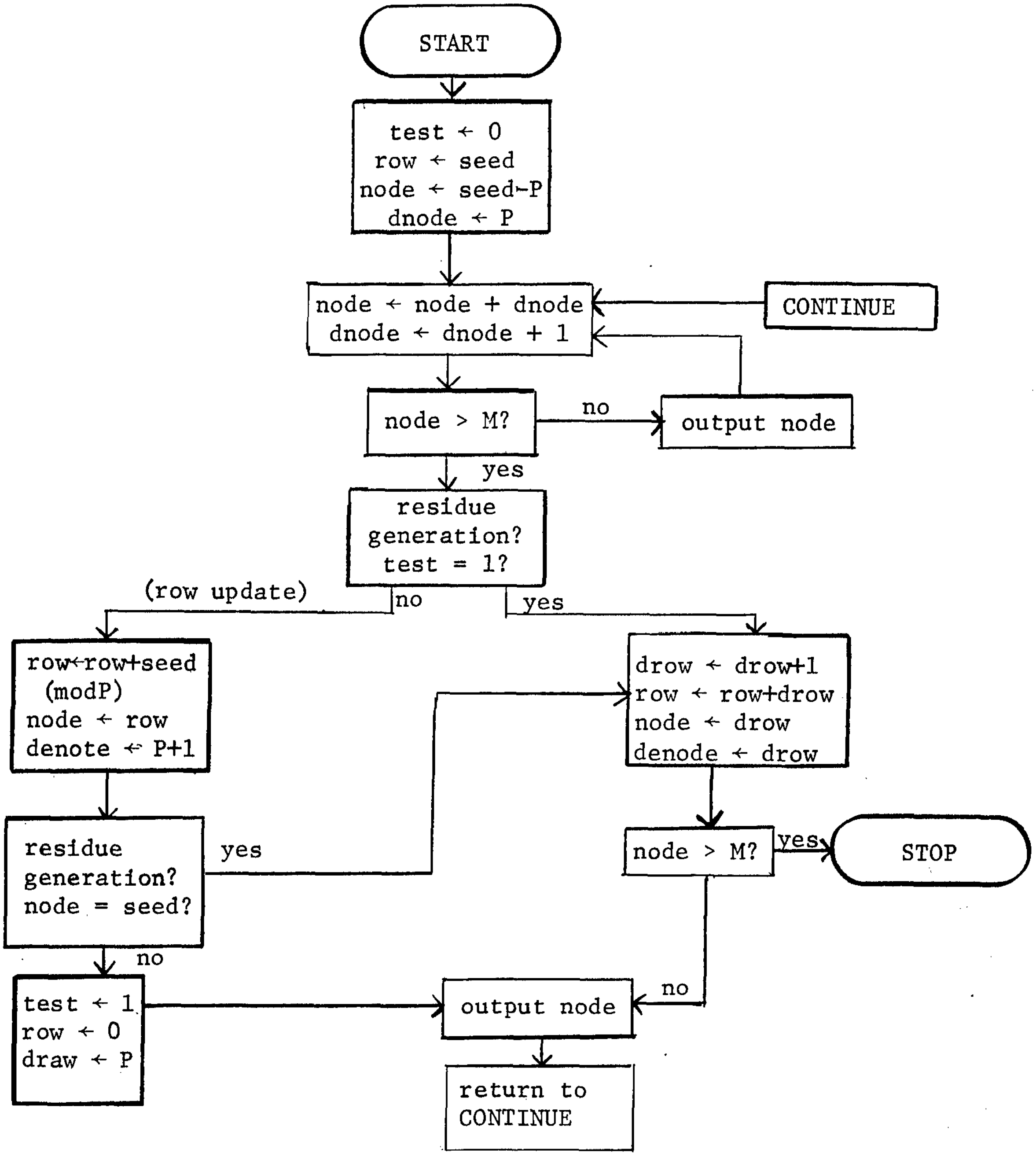


Figure 10 - Node Generation Algorithm

The probability of hitting a head (or last) cell is

$$p(H) = \frac{1}{K} \sum_{y=1}^{M} (1 - p_o(y)) = \frac{M}{K} - \frac{1}{K} \sum_{y=1}^{M} p_o(y) \tag{6}$$

With these probabilities we now calculate the expected conflict list lengths for FIND$^-$, FIND$^+$, ADD applied to a hashed address $x \doteq T(A,B)$.

$$E[\text{FIND}^-] = 1 \cdot \frac{1}{M} \sum_{y=1}^{M} p_o(y) + \frac{1}{M} \sum_{k=1}^{K} k \sum_{y=1}^{M} p_k(y) \simeq \frac{1}{M} \sum_{y=1}^{M} e^{-pk} + \frac{K}{M} \tag{7}$$

$$E[\text{FIND}^+] = \sum_{k=1}^{K} \left(\frac{k+1}{2}\right) p(K) \simeq 1 + \frac{K}{2} \sum_{y=1}^{M} p^2(y) \tag{8}$$

$$\begin{aligned} E[\text{ADD}] &= 4p(x \in AS,H) + 6p(x \in WS,H) + 7p(x \in WS,-H) \\ &= 4\frac{(M-K)p(H)}{M-K(1-p(H))} + 6\,\frac{K}{M}\,p(H) + 7\,\frac{K}{M}\,(1-p(H)) \end{aligned} \tag{9}$$

Moreover, the variances of these operations are

$$\text{var}[\text{FIND}^-] = \frac{1}{M} \sum_{y=1}^{M} (e^{-pk} + p^2 K^2) + \frac{K}{M} - (E[\text{FIND}^-])^2 \tag{10}$$

$$\text{var}[\text{FIND}^+] = \sum_{j=1}^{M} \left(\frac{3}{2}p + \frac{7}{4}p^2 K + \frac{1}{4}p^3 K^2\right) - (E[\text{FIND}^+])^2 \tag{11}$$

$$\text{var}[\text{ADD}] = 16\,\frac{(M-K)p(H)}{M-K(1-p(H))} + 36\,\frac{K}{M}\,p(H) + 49\,\frac{K}{M}(1-p(H)) - (E[\text{ADD}])^2 \tag{12}$$

As an example, plots are given for $E[\text{FIND}^+]$ and $\left\{E[\text{FIND}^+] + \sqrt{\text{var}[\text{FIND}^+]}\right\}$ for $K = 50$, $M = 100$ in Figure 11. Since p(y) is not in general unimodal and symmetric, the plots for the standard deviation should be regarded with caution as giving a rough idea of the spread of accesses. One could with some inconvenience, discount the assymetry of p(y) by calculating right and left deviations, but the curves in Figure 11 represent the mean plus right standard deviation to within one access, and, in any event, suffice to display the proportionality of spread between the various cases. The best and worse cases were calculated by successively generating nodes whose distribution minimized and maximized $E[\text{FIND}^+]$, respectively. As predicted in the discussion of Equation (1), the curves in Figure 11 show that for $m/M \geq .5$ the number of accesses required is unaffected by the generating method. Moreover, for $m/M \leq .5$, the number of accesses required in the worst case can be two to three times as great as for the best case. Indeed, a glance at Equation (8) indicates that since the squared emphasis on clusters is not degraded for increased K, the number of accesses for fixed m can rise proportional to K. More important, Equation (9) indicates that, under the same conditions, the standard deviation also increases monotonically with K. Since the choice of the prime clearly can approximate either the best or worst cases, we must now determine a method for selecting the prime.

From the row generating sequence of Equation (2), one can specify the m for which the m^{th} generated node in the row is less than or equal to the modulus M.

$$s_j + \frac{m^2}{2} + (P - 1/2)\, m - P \leq M \qquad (13)$$

With no loss of generality, one may choose s_j=P. Then, for equality in Equation (13), one finds:

$$\hat{m} = -(P - 1/2) \pm \sqrt{(P - 1/2)^2 + 2M} \qquad (14)$$

Since $m \leq M/2$ specifies the limit of our concern with non-clustering, we can fix the number of residues at M/2 by the equation

$$\hat{m}\, P = M/2 \qquad (15)$$

Thus, by Equations (13) and (14), one has

$$P \simeq 1/2 \sqrt{M} \qquad (16)$$

Indeed, for the example in Figure 11, $P = 1/2 \sqrt{100} = 5$ is very close to the "best" case.

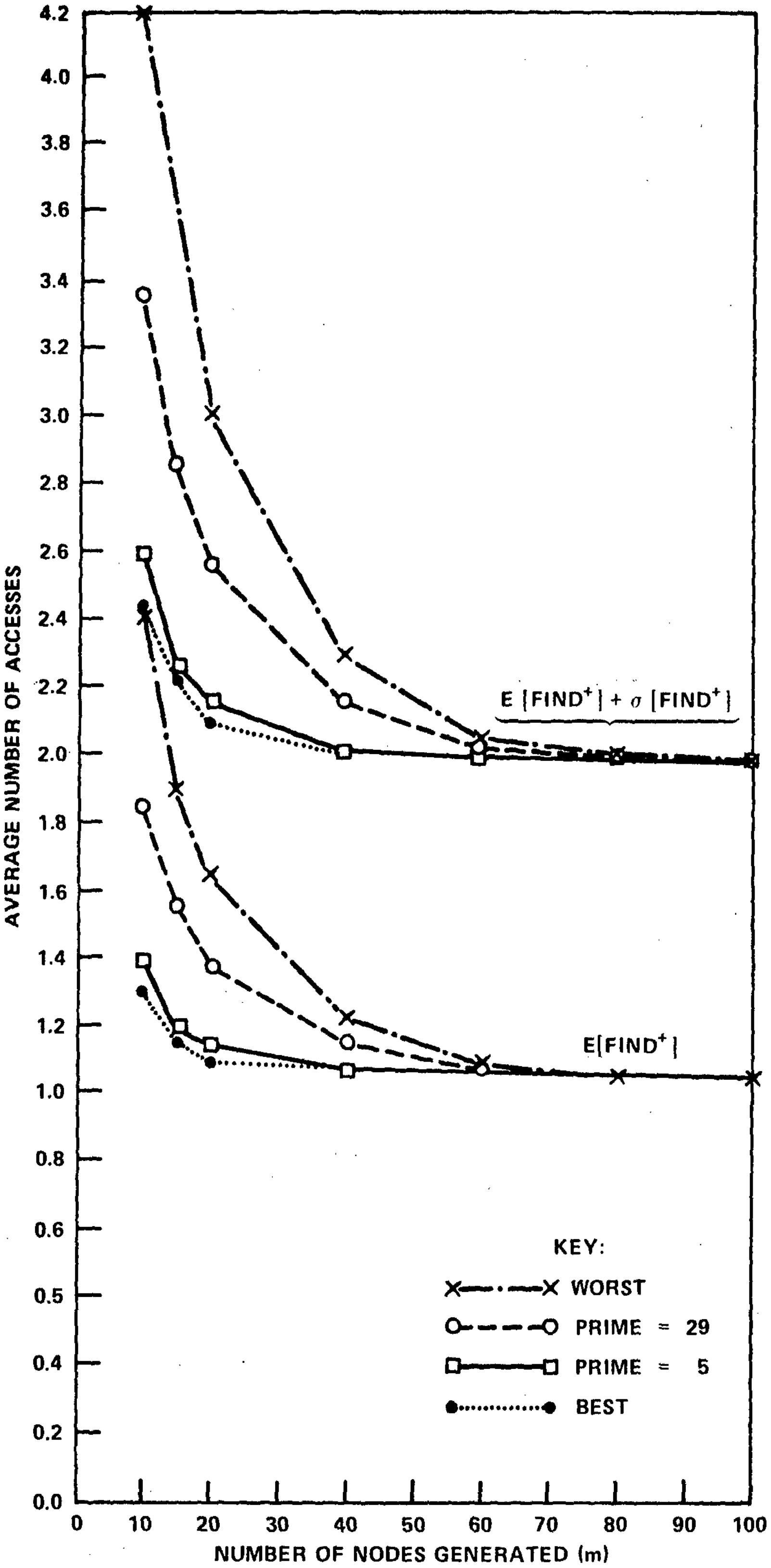


Figure 11 – Expectation and Standard Deviation of (Find $^+$) versus Number of Nodes Generated (K/M = 0.5)

Memory and Timing Considerations

As in design of all data management schemes, there has been a tradeoff in GIRS between various competing criteria. In order to achieve high speed and structural flexibility, we have introduced direct chaining at the expense of a 50 percent increase in memory requirements and considerable executive complexity. On the other hand, Equation (8) shows that the average number of accesses is low (at best, less than 1.5) for any percentage of memory filled. Therefore, it is economical to run with the memory as full as desired. To some extent, this fact balances the amount of memory needed, since many other data management schemes cannot run profitably with full memories.

The availability of a Sequential Space offers a balance between the dynamic flexibility of a list-processing memory allocation scheme and the rigid but rapid response of matrix storage defined at execution time. In passing, one might note the further availability of pointing to arbitrary matrices defined at compile time. If R is DIMENSIONed in labelled COMMON, then one can point to R by making the value of LOC(R) an integer datum sink node of some triple. When LOC(R) is retrieved and set equal to some J, then the Ith location of R is found by accessing J(J-LOC(J)+I).

One should not take the above to mean that GIRS is suitable for all data management applications. For example, in LISP[6] the expression ((M.N) X (M.N)) takes a minimum of four cells, whereas GIRS would require at least six. On the other hand, the LISP association ((A.B)C)) or (A B C) requires three cells, whereas GIRS requires one. The distribution is rooted in the fundamental design of GIRS to be an <u>associative</u> memory, whereas LISP, from a graph-structural viewpoint, contains only either non-terminal nodes with unary or binary fan-out in homogeneous (i.e., unlabelled) linkage or a terminal list of two nodes. Thus, LISP handles nested pairs of objects quite efficiently due to the (tacit) use of binary homogeneous links, but does not fare so well with associations.

The GIRS design has produced the average retrieval, insertion, and deletion times on the CDC 6700, as shown in Table 1. As expected, the insertion times exceed the retrieval times, but not significantly. Considering the linkage sophistication and the fact that GIRS is written in FORTRAN EXTENDED with no (by hand) machine language optimization attempted, the times seem quite reasonable. On the other hand, one should note that each field was assigned a separate word (as might be done in a byte-oriented computer), and the packed version (one word per cell) will run somewhat more slowly. Note also the saved index feature mentioned previously which leaves the time to search down a list uniform for each access.

TABLE 1 - AVERAGE INSERTION, DELETION, RETRIEVAL TIMES

Time (microseconds)	Action
59.4	INSERT - univalue, no conflict
82.8	INSERT - univalue, one conflict
116.5	INSERT - 2nd value, no conflict
116.4	INSERT - 3rd value, no conflict
60.6	DELETE - univalue, no conflict
88.3	DELETE - 2-value list, no conflict
27.7	FIND - triple absent
51.2	FIND - univalue, one conflict
68.7	FIND - 1st on multivalue list, no conflict
70.7	FIND - 2nd on multivalue list, no conflict
71.0	FIND - 5th on multivalue list, no conflict

OUT-CORE STORAGE DESIGN

When an in-core graph overflows the memory, one must have a provision for storing and accessing the excess data in an external file. The primary objective of the file design is to minimize the number of disk reads per graph association access. The objective is more difficult to realize for arbitrary graphs--i.e., highly structured data--than for loosely structured blocks of sequential data for which the ratio of relationships (points, associations) to data words is much less than unity. The problems associated with an effective partition of a graph fall into three categories:

- local graph processing
- page reorganization
- generalized inversion

which we now discuss.

Local Graph Processing

The handling of overflow as an immediate global extension of in-core storage--e.g., by means of scatter index table, as discussed in Morris[1] is not suitable for our purposes since it requires one disk read per graph access. Instead, we opt for a partition of the memory into pages each of which permits a local massage of the graph by an analyzing program. This is considerably more difficult than it sounds. Basically, the local character of a page subgraph is a matter of interpretation by the analyzing program, and hence must be left to some extent in the analyst's hands. It seems quite clear that unless software can be developed to understand a block of code with respect to some universe of logically structured data--a difficult, sometimes impossible task for humans--the task of efficiently structuring the data for a particular implementation and execution can never be fully automatic. On the other hand, some insight into the way graphs are generally manipulated may facilitate the formulation of reasonable partition-constructing tools for the user.

As we have mentioned, any graph may be decomposed into primitive structures called node-link-node triples. Certain more complex structures of these primitives may generate a higher level decomposition whose resolution decreases as the decomposing structure becomes more complicated. One fundamental such decomposition maps a graph into *lists*--a sequence of triples all containing the same source node and link; and into *strings*--a sequence of triples each constrained to have its sink node identical with the source node of the succeeding triple. This decomposition is, in a sense, both *arbitrary*--why not choose lists of length greater than two or choose trees and circuits--and *redundant*-- cannot any graph implemented by lists and pointers also be implemented by strings, and vice versa, however awkwardly? Indeed, in a list the indices are implicit in the implementation, whereas in a string the nodes--aside from a superfluous interpretation as, say, automaton states--serve as explicit indices in a link sequence. The choice of a *combination* of lists and strings is directly related to the conventional use of graphs both in the forms of generative directories, stacks, and sets--logically included in a list representation--and in the forms of recognitive directories, trees, and circuits--logically represented by strings. Similarly, the *distinction* between lists and strings is made not only for the sake of the logical representations just mentioned, but also to take advantage of the underlying memory structure of in-core

GIRS which permits, for example, a non-imbedded string--one whose nodes have no more than one link apiece, i.e., a _chain_--to be most efficiently represented by a list.

In order to get a fix on the problem of graph localness using the list/string decomposition, we first posit the skeleton structure of a page. The _body_ of a page has exactly the structure of in-core GIRS and is partitioned between WS and AS. (SS is handled as a straightforward extension of in-core SS.) The address of a cell in the body has two parts: a page number _prefix_ and a cell number _suffix_. Hashing is performed internal to a page on the suffixes of the source node and link, modulo the body length. The _header_ of the page contains information about the page, such as the current cell entry to AS and the last node generated; other header cells will be detailed later.

The central problem is to determine on which page one should place a newly created node. If the program is creating a list of names of major graph localities, each new node should be on a different page than the current one, since, in the skeleton structure outlined above, each page will have the capacity to contain at least the beginning of a graph locality. On the other hand, if the list is part of a hierarchical directory, then only the terminal nodes of the directory should be on different pages than the current one. Similarly, if the program is setting up strings to operate in a trace mode, then the newly created sink nodes should be on the current page or at worst on few enough other pages so that primary memory could contain all of them. More generally, graph manipulations are, in part, combinations of directory searches and string traces. While a directory search can result in a separate disk access to a new graph locality, it is most desirable that a trace take place within a given locality on one page. For example, consider a stack search which provides the information to continue a trace. In this case, the stack nodes are best represented as a list--i.e., a terminal directory--on a single page. The sink node addresses however, are intended to represent links in a single graph locality, and hence are all used on the same page or at least on some one other page. If the link addresses are never used as source nodes, their place of definition is irrelevant.

In order to implement strategies of the above sort, manually or automatically, we need to establish a mechanism not only for creating a node--the suffix of the node address- but also for stipulating the page on which the node address is to reside--the prefix of the node address. The operations are independent so that a node can be created at one time with a zero prefix and placed on a page when the program context demands it. Since the AS of each page is different, the initial node address definition is a temporary address and will be fixed when assigned to a page. In default of either a direct page stipulation or embedded strategy, the node must be placed on the in-core page last accessed, the <u>current page</u>. In order to save access time, we presume that our insertion strategy has localized the search, and we begin all searches on the current page. If that fails, we search the other in-core pages. If that fails, we only then seek a new page from disk. Hence, we also provide an <u>in-core page directory</u> which lists the pages and their in-core locations and which contains an indicator cell that specifies the current page. Since we intend to embed GIRS in another paging scheme--e.g, the CDC Data Handler, or some variation thereof--it is not necessary to prescribe the control memory and operations (like create, destroy, access, copy a page) necessary to efficiently manipulate the pages. Here we are only interested in the tools necessary to effect a strategy to choose pages for newly created nodes. This is effected by having the INSERT operation test the elements of the candidate triple for zero prefix, an indicator of definedness. If a prefix is a zero, INSERT will call a STRATEGY procedure to be supplied by the user. A typical STRATEGY might be: "If the link is A, place the sink node on page 3; if the link is B, place the sink node on page 4; otherwise, default." Such a strategy might be coded in terms of GIRS itself. The additional time required to execute the STRATEGY in INSERT would be saved many times over through the reduced number of disk reads. If the source node prefix is non-zero, the triple is inserted on the page denoted by that prefix.

In order to achieve a successful partition, one might conceivably adopt a strategy that would place the same source node on two or more different pages. An instance of this sort might be one in which a single node served as entry to two quite independent graphs, or better, one in which a single graph is partitioned into two on the basis of a partition of the link set. Aside from the minor difficulty of having to redefine nodes of non-zero prefix, the major resultant problem would be that a table of address equivalences might have to be established, queried at every access, and thinned out when certain deletions occur. Moreover, the table (or tree) would potentially contain as many cells as all of memory. On the other hand, by restricting the size of the table and flagging those nodes with addresses on several pages, we can introduce address equivalences. Never-the less, we have achieved this effective indirect addressing by the use of flags which must be tested at every access, and it seems that the same effect could sometimes be achieved at less cost by properly restructuring the graph. However, there is a solution which entirely avoids the use of tables, as we discuss in that which follows.

In order to account for the possibility of a graph subset overflowing a page, one might be tempted to resort to variable length pages. However, not only is such a device notoriously difficult to implement, but in fact it would not serve the purpose, since such a page might overflow in-core memory. Rather, we introduce into each page header a continuant cell which contains a pointer to the next page - called a continuant - containing the overflow. Thus, a page effectively consists of a sequence of continuants. The last such continuant contains a zero continuant cell. Since one may have to get back to the head of a continuant chain,--a reverse continuant cell containing the head continuant address must also be located in the header. The first reverse continuant cell is zero. Thus, overflow pages are linked by an up-down list. In addition, since all continuants are referrent-facsimiles of the head continuant and thus have the same page number, the in-core page directory must indicate the sequence index of the continuant so that an entry is available to a page search. Since all continuants have the same page number, the node addresses of the head continuant are the same as those of any continuant; therefore, the continuants offer an immediate alternative to the address equivalence problem noted above. In order to implement such a solution, however, one must introduce the capability of specifying not only the page on which a triplet is to be inserted--as given by the source node prefix--but also, as an option, the continuant.

Page Reorganization

At some point in processing a graph, it may become desirable to reorganize the continuants of some page for one of two reasons: (1) either because the current graph structure is inefficiently distributed over the pages of memory--a formal inefficiency; or (2) because the processing program requires a different partition--a content-oriented inefficiency.

A prime example of a formal inefficiency is the spreading of a multivalue list among several continuants, thus increasing the time required to traverse the list. In order to use a post-facto reorganization procedure, a count would have to be made of each insertion which places a triple on a continuant as a result of an overflow. Consequently, each header would contain a multivalue overflow cell which would count the number of such insertions. If any access resulted in the count (on all the continuants accessed) exceeding a threshold, a multivalue list reorganization would be performed which synthesized a multivalue list on one page. In order to make room for entire multivalue lists on a single page a new continuant might have to be formed. Not only is the foregoing procedure awkward and time-consuming, but one must also consider the inefficiency of accessing several pages just for the elements of one multivalue list. Clearly, the solution is to perform a dynamic reorganization by moving a multivalue list that would overflow a continuant to a continuant with sufficient capacity.

If the list is too long for any continuant, then despite the suggestion that the page length should have been increased, we must make allowance for a multivalue list continuation pointer in the last list cell, indicating the continuant holding the rest of the list, and a corresponding multivalue list continuation flag indicating that the last cell holds a pointer. Thus, average insertion time is increased, but average access time is kept at a minimum, which is acceptable since we expect many more reads than writes. Nonetheless, one must realize that average access time for the out-core design is bound to be more than that of the in-core memory since the continuants have to be searched. In particular, for attempting to access an item not in the store, all the continuants must be searched; thus FIND$^-$ could have the longest access time--whereas in the in-core design, it had the shortest--unless the page size is judiciously chosen.

An example of a content-oriented inefficiency occurs when a partition is needed which places the same source node on two or more different continuants. Clearly, there is no way to decide on a formal basis whether the separation is advantageous or not. Moreover, if an arbitrary continuant were permitted to be chosen for the insertion of a triple, then some reconciliation is necessary between this freedom and the constraint of keeping multivalue lists on a single continuant as prescribed in the preceding paragraph. An obvious, reasonable, and inescapably awkward way out is to permit source node separation only when the links are different. A direct result of this policy in particular and of node separation in general is that both when inserting (INSERT) a triple on a particular continuant and when searching for a triple not in the store (FIND$^-$) all the continuants must be searched, even if the appropriate conflict list was found on an early continuant. A partial alternative to this search procedure might be to put a continuant specification option in FIND--as was done before for INSERT--and to require that whenever the option was invoked, the search be confined to the continuant specified. There is a danger, of course, that a multivalue reorganization will have moved the triple in question unless we dictate the overriding of any such reorganization upon the use of a continuant specification option during insertion, and institute a flag, the REORG flag, to inhibit any further reorganization. Advantage is then taken of the multivalue list continuation pointer and flag as described earlier for lists which overflow the continuant. The decision as to whether a source node may appear on more than one continuant should be possible not only through the continuation specification option, but as an embedded constraint in the STRATEGY procedure. On the other hand, post-facto reorganization--which might be required, for example, to merge a subgraph constructed on a tentative, separated basis with a more permanent structure--must be done consciously. Nevertheless, we will provide procedures MERGE--to merge two continuants--and SEPARATE--to move a multivalue list specified by source node and link to a given continuant.

Generalized Inversion

Up to this point, we have discussed intra-page, rather than inter-page reorganization. Now we broaden our scope. The kind of question that has been posed to the memory can be symbolized in the form

$$A\ B\ ? \tag{17}$$

That is, "What is the object(s) associated with a given source object by a given attribute?" The usual inverted interrogation has the form

$$?\ B\ C \tag{18}$$

That is, "What is the source object(s) which are associated to a given sink object by a given attribute?" The structure of GIRS posed so far does not yet reflect the necessity to answer questions of this inverted sort, nor indeed of the following sorts, which are obvious variations of the above.

$$A\ ?\ C \tag{19}$$

$$?\ ?\ C \tag{20}$$

$$?\ B\ ? \tag{21}$$

$$A\ ?\ ? \tag{22}$$

$$?\ ?\ ? \tag{23}$$

For inversion in a more general sense, one may declare any collection of partly bound triples to be primitive and allow the memory to be interrogated on the unbound portions. For example, in Figure 12, the graph primitive has the insertion schema

$$A\ (B\ C,\ D\ B(F\ G,\ H\ A)) \tag{24}$$

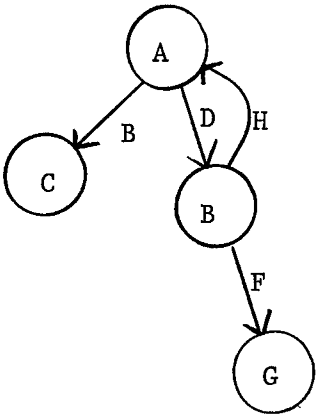


Figure 12 - Example of Graph Primitive

An example of an appropriate question might be

$$A(X\ ?,\ E\ X\ (?\ S,\ H\ A)) \tag{25}$$

where A, X, E, S, H are bound. Unlike the questions above, we have introduced a labelling constraint into the primitive, so that implementations of a schema may be required to have identical substitutions for the bound variables. Thus, the following question is illegal:

$$A(X\ ?,\ E\ X\ (?\ S,\ H\ Z)) \tag{26}$$

unless Z = A, and the following question is irrelevant:

$$A(X\ Y,\ E\ X\ (R\ S,\ H\ ?)) \tag{27}$$

since ? is bound to be A. Moreover, the free variable ? must sometimes be restricted, as in the following question:

$$?_A\ (X\ ?,\ E\ X\ (?\ S,\ H\ ?_A)) \tag{28}$$

It seems reasonable to preface the question of whether or not a memory scheme can be devised to efficiently contain an arbitrary graph primitive with the question of whether it can be done at all. Since a primitive schema is a set of triples, there is clearly a solution if one can handle the inversion problem for triples alone. Indeed, as we shall see, there is a range of solutions which determine a trade-off between the complexity of memory and the dynamic flexibility of graph primitives.

For triples inversion, a solution has been suggested by Feldman and Rovner[2] in describing the memory scheme for the LEAP language. In order to answer the single "single-question-mark" queries of (17), (18), and (19), triples are inserted by hashing not only (A, B), but also (A, C) and (B, C), and presumably setting a flag to indicate the query type. There are some important detailed differences between the LEAP and GIRS organizations, but the effects are similar. The "double-question-mark" queries of (20), (21), and (22) are satisfied by introducing a new field into each triple which links all the triples having the same hash suffix for the query type under consideration-i.e., the B of "? B ?" or "? ? B". The triple-question-mark query (23) implies accessing of all memory and is not relevant.

In principle, the triples inversion scheme can be directly extended in several ways to more general schema such as the schema (20). Any schema consists of say n distinct components X_i (i=1, . . ., n) described by the n-tuple $\underline{X} = (X_1, ..., X_n)$. A sequence suffices since the form of the graph-i.e., the parenthesization of the schema - can reside in a table for common reference by any instance of the schema. Any query is a partition of $\underline{X}$ into the given information $\underline{Y}$ - on which the key is based - and the missing information $\underline{Z}$ - the question-mark/components. A transformation $T(\underline{Y})$ provides the address at which a cell of n fields (excluding the flag field) is located, as opposed to the triples cell of three fields. In terms of triples as primitives, the n component schema would have required at least $3(1 + (n-3)/2) = 1.5(n-1)$ fields (where n is odd and greater than one). One could implement inversion either by redundant insertions of $\underline{X}$ by means of all the various subvectors $\underline{Y}$ regarded as keys; or by introducing additional inversion linkage fields to chain all $\underline{Y}$ containing a common subvector. However, such an implementation requires consideration of 2^n-2 partitions of $\underline{X}$ and leads us to the following conclusions:

- In general, complete inversion for even moderate values of n is not feasible, because a memory of combinatorial proportions is required, and also since an enormous amount of time is required to continuously update the memory.

- Partial inversion--the specification of a small fixed set of $\underline{Y}$'s to be used as keys for redundant insertion of $\underline{X}$ or as inversion links--is feasible only for a system permitting rigid forms of query on a moderately stable data base. The easiest way to implement a partial inversion would be a redundant insertion into SS keyed on some $\underline{Y}$ with a pointer to a prototype structural description elsewhere in SS.

- For a more flexible query format and a more dynamic data base, a somewhat inefficient compromise is to allow the query pattern to be stipulated at a higher level for each insertion as described for the Pattern Information Retrieval Language PIRL[3]. The pattern--i.e., subgraph--is based on the simultaneous storing of n-tuple and structure in SS. Partial inversion is facilitated by declaring a list of source nodes and/or links on the basis of which a LEAP-like inversion structure of primitive triples is inserted. Retrieval inversion of the pattern is handled through a sequence of queries on primitive triples, in a LEAP-like structure. The principal advantage of this option is that hierarchies of subgraphs can be described, named, and manipulated much the same as nodes in the ground level graph. The main disadvantages are the proliferation of subgraph structural description with the attendant memory cost and the possible many disk reads necessary to access a sequence of triples on the basis of an inter-page spread of source nodes.

Perhaps the ultimate dilemma facing the designer in any of the proposed memory allocation schemes is the problem of inter-page reorganization. As long as queries must be answered by an immediate direct access to memory, then the re-partition of schemata for a new form of query demands a complete cell-by-cell overhaul of memory both to add the new retrieval form and possibly also to delete the old form. This is clearly a tedious procedure and is not likely to be acceptable for flexible and dynamic forms of query. On the other hand, one can relieve the formal storage rigidity of the demand for a direct, one-to-one retrieval access per query by allowing the FIND procedure to search for the answer to a query by using knowledge of the types of permissible associations in memory. Such an inferential capability may be found in TRAMP[4] for example. Here, however, we will not stipulate an underlying search mechanism, but will rather give the user the opportunity to imbed an explicit strategy in the procedure INFER which will be used by FIND. INFER is a device which can lend great flexibility to the range of queries and therefore sidestep the need for re-organization; all at the cost of a potentially long and unguaranteed search. Moreover, one can specify INFER on several levels. For example, a system programmer might detail the explicit conditions under which a graph is to be searched, depending on the query format, and merely modify INFER if the query format is changed. On the other hand, a somewhat more static arrangement may be offered to the applications programmer: for example, he might be required only to submit a tree of associations demanded by a fixed, imbedded strategy. A more detailed example is now in order. Consider the data tree structure in Figure 13 on the following page.

```
fleet
     (ship
         (type,
          hullno,
          dimens
               (length, beam, height),
          displacement
               (empty, gross),
          structures
               (deck
                        (id, weight, xcoord),
               frame
                        (id, weight, xcoord),
               bulkhd
                        (id, weight, xcoord))))
```

Figure 13 - Ship Design Data Structure

The levels of the tree are given by the nesting levels of the parenthesization. The tree is in fact a *relationship graph* of links and would be logically represented in the computer as a set of nodes connected by UP and DOWN links. For example, *deck* would have a UP link to *structures* and a DOWN link to *weight*. Any name in the data structure of Figure 13 which immediately precedes a comma or right parenthesis is a terminal link which, in the actual data base graph, points either directly to data or indirectly (via an index) to a matrix location holding the data. Any nest of terminal links may in fact be replicated many times in the data base. The queries that we would expect to answer are of the following kinds.

- Direct, memory-related - that is, those queries answerable by a single access to memory, such as

  What structures does (ship) A have?

  What types of structures do ships have?

Note that the first query is to the data base, whereas the second is to the relationship graph.

- Direct, inverted - that is, those queries which would require a single access if the inversion relation exists in memory, otherwise a search from a given entry node is required. For example,

  What objects have the structure A?

  The entry node for an inversion search would be <u>fleet</u>.

- Indirect, memory-related - that is, those queries requiring a chain of links to secure the answer. For example,

  Which decks does (ship) A have?

  Which decks do ships have?

- Indirect, inverted - such as,

  Which ships have the structure C?

  Which objects have the structure C?

  Which ships have the deck C?

  Which objects have the deck C?

  How are the (ship) B and the (deck) C related?

- Direct or indirect, memory-related and inverted - For example,

  What objects are related by a structure type?

  (That is, for which objects A, B, is B a structure of A)

  Which objects are related to deck A?

In Figure 14, Page 43, the flow chart for the search required to answer an indirect, memory-related query is shown. The GIRL/FORTRAN[4] code is given in the Appendix. Searches for other queries are quite similar and will not be given here since, as will be discussed, the inference schemes will vary from data base to data base. That is to say, at the level of interest (triples retrieval) it would be inefficient to stipulate a general inference procedure for all data bases since each data base may be associated with a peculiar set of heuristics to enhance the search. For example, instead of trees, one may wish to deal with graphs containing circuits; or to build inversion links or forward links; or to remove them either in the

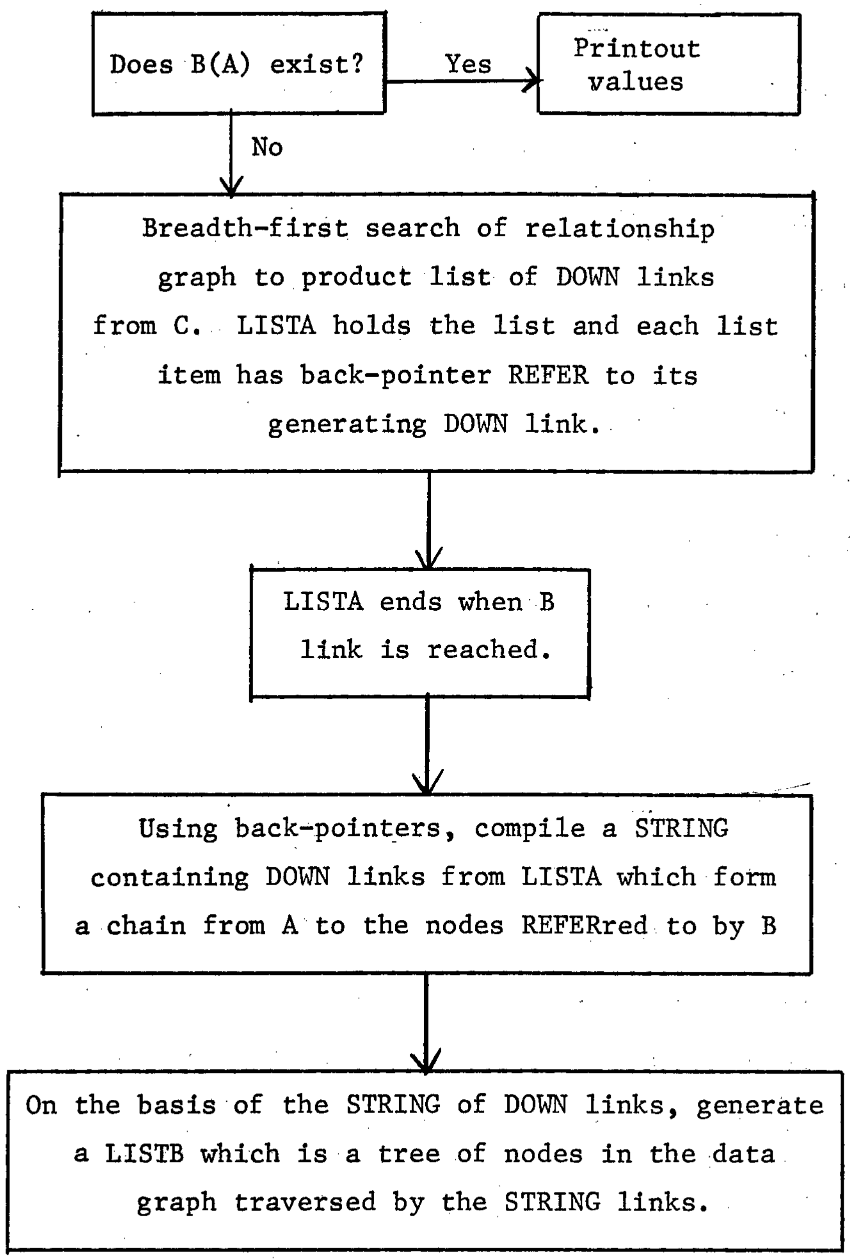

Does B(A) exist?
Yes
Printout values
No
Breadth-first search of relationship graph to product list of DOWN links from C. LISTA holds the list and each list item has back-pointer REFER to its generating DOWN link.
LISTA ends when B link is reached.
Using back-pointers, compile a STRING containing DOWN links from LISTA which form a chain from A to the nodes REFERred to by B
On the basis of the STRING of DOWN links, generate a LISTB which is a tree of nodes in the data graph traversed by the STRING links.

Figure 14 - Diagram of the Search Procedure for an Indirect, Memory-Related Query

relationship graph or in the data base on the basis of frequency of usage; a kind of partial inversion. In fact, the user may wish to partially specify the inversion links beforehand. In a like vein, for retrieval systems which process relatively few queries per user, an interactive approach might be used so that the system would be directed in its search by the user. Moreover, the breadth-first, top-down search given in the example is both guaranteed and blind. Many other search techniques exist--not so blind but not so well guaranteed--as, for example, those discussed by Nilsson[7]. On the other hand, at a somewhat higher level of discourse it may be more worthwhile to consider a fixed search strategy. Such a strategy is describable, for example, in PIRL[3] which is designed to handle sequences, logical conjunction, disjunction, complementation, and quantification of triples and schemata of triples. The basic lesson to be derived from the example is that an inference search requires two elements: (1) an abstract, heuristic view of the data base structure and its interface with a given class of query formats (in the example, a relationship graph and a method of searching it); (2) the data base itself (in the example, a simple hierarchical tree terminating in numerical data). By formally abstracting the attribute relationships from the data base, we are able to avoid implanting the relationships in a redundant fashion within memory. In trade for this economy in storage, we are obliged to engage in a content-oriented search which may be both expensive and unsuccessful. In a sense, we have introduced a range of hierarchies of forgetfulness into the memory in exchange for a contraction of memory, and hence less paging.

Method of Operation

Before describing the operations, we briefly summarize the memory structure suggested during parts of the preceding design discussion. In-core memory contains (1) locations for a set of uniform length page continuants drawn from secondary storage and (2) an in-core page directory. Each continuant has a head and a body. The head contains four cells: the entry to AS, the last node generated, and pointers to the preceding and succeeding continuants. A page consists of a set of continuants arranged in an up-down list, with the pointer of the first continuant and the reverse pointer of the last continuant being zero. The body of a continuant is arranged as for the in-core design, except that each address has a prefix which is the page number. The set of addresses of the continuant of a given page is the same set as that for any continuant of the same page. The last cell of a multivalue list on a given continuant may be a pointer to more of the same list on another continuant only if Flag 11 (defined to be the multivalue list continuation flag) is set. If the multivalue list is not eligible for translation to another continuant, Flag 12 (the REORG flag) is set in the head cell. Moreover, setting Flag 13 (the list continuation flag) in the head cell of a multivalue list indicates that the list is a non-movable continuation of a list on some other continuant. The in-core page directory contains (1) the current page cell and (2) a vector of (page number, continuant number) couples as a function of their current location in core.

Retrieval: Find B of A, possibly given a continuant and/or entry (to INFER).

Step 1. If the continuant of the A-prefix page is specified, fetch the continuant and apply FIND(A,B).

Step 2. Otherwise, fetch each continuant of the A-prefix page and apply FIND(A,B) to each in turn, remembering to disregard head cells with Flag 13 set.

Step 3. If FIND(A,B) is not successful on any continuant then call INFER (possibly with entry, if supplied).

Insertion: Insert the link B from A to C, possibly given a continuant D and/or entry to STRATEGY.

Step 1. If D of the A-prefix page is specified, FIND(A,B) on D.

a. If the B(A) list is complete on D and there is an available cell, then INSERT(A,B,C) on D with STRATEGY and set F12 of the head cell. If there is no available cell on D, then fetch the next continuant D' that has three available cells. Replace the last sink node C' on D-list by a pointer to D'. Set F11 on the last cell of the D-list. INSERT(A,B,C') and (A,B,C) on D' with STRATEGY--remembering to carry along the flags of C'--and set F12, F13 on head cell of D'-list.

b. If the B(A) list is not complete on D, then follow the multi-value list continuation pointers until a continuant D holding the end of the list is reached. Complete the insertion of (A,B,C) as in 1a.

Step 2. If the continuant D of the A-prefix page is specified, but FIND(A,B) does not succeed on D or anywhere else, then perform the insertion (A,B,C) as in 1a. However, if FIND(A,B) succeeds on some D' and F12 does not inhibit a reorganization, then move the list from D' to D and complete the insertion of (A,B,C) as in 1a. If F12 is set, however, an error is reported out; nevertheless, insertion takes place on D' instead of D as in 1a.

Step 3. If no continuant of the A-prefix page is specified then seek a continuant D on which FIND(A,B) succeeds and complete the insertion (A,B,C) on D as in 1a, except that F12 is not set. If no continuant is found on which FIND(A,B) succeeds, fetch any continuant with one available cell and INSERT with STRATEGY.

Deletion: The deletion procedure is a straightforward FIND and trace through continuants to return all of a multivalue list to AS, and will not not be detailed here further.

IMPLEMENTATION

GIRS has been implemented in FORTRAN EXTENDED on the CDC 6700 by I. Zaritsky. The implementation is based on earlier work performed by A. Bandurski and P. Friedenberg. A User's Manual will be published shortly.

APPENDIX A

SEARCH PROCEDURE FOR INDIRECT, MEMORY-RELATED QUERY--WRITTEN IN GIRL/FORTRAN

INFER: B of A given that A is of type C. If (C=0) RETURN

```
C   ***   LISTA HOLDS DOWN LINKS OF C
G    2    LISTA(HOLDS C 'TEMP, REFER TOP)
          JTEMP=0
     3    J=0
G    4    TEMP + DOWN/6 .'J=J+1'/6=B/'NEXT 5/7
G    5    LISTA(HOLDS NEXT, REFER 'JTEMP'//4)
G    6    LISTA(+HOLDS.'JTEMP←JTEMP+1' /15/'TEMP3)
C   ***   COMPILE STRING OF LINKS FROM C DOWN TO B
G    7    LISTA STRING(B,TEMP)
G    8    LISTA(+REFER.JTEMP'JTEMP=TOP//9,+HOLDS.JTEMP'TEMP,STRING
          TEMP//8)
C   ***   GENERATE LISTB:TREE OF NODES BASED ON LINK STRING
     9    J=0
G         A'NODE
G   10    LISTA+STRING.'J=J-1''TEMP=B//13
          K=0
G   11    NODE+TEMP.'K=K+1'/12'NEXT
          LISTB(STRING NEXT,REFER 'J'//11)
G   12    LISTB(+STRING/16(.1'NODE,-.1,+REFER(.1'J,-.1//10))
C   ***   OUTPUT NODES LINKED BY B
    13    JJ=0
G   14    NODE+B.'JJ=JJ+1'/12'OUT
          PRINT(OUT)
          GO TO 14
    15    PRINT('FAIL')
    16    RETURN
          END
```

REFERENCES

1. Morris, R., "Scatter Storage Techniques," Communications of the ACM, Vol. II, No. 1, pp. 38-44 (Jan 1968).

2. Feldman, J., and P. Rovner, "An Algol-Based Associative Language," Communications of the ACM, Vol. XII, No. 8, pp. 439-449 (Aug 1969).

3. Berkowitz, S., "PIRL--Pattern Information Retrieval Language--Design of Syntax," Proceedings of the 26th National Conference of the ACM, pp. 496-507 (1971).

4. Berkowitz, S., "GIRL--Graph Information Retrieval Language--Design of Syntax," Software Engineering, Proceedings of the COINS--III Conference, Edited by J. Tou, Academic Press, Vol. 2, pp. 119-139, (1971).

5. Ash, W., and E. Sibley, "TRAMP, An Interpretive Associative Processor With Deductive Capabilities," Proceedings 23rd National Conference of the ACM, pp. 143-156, (1968).

6. McCarthy, J., et al., "LISP 1.5 Programming Manual," MIT Press, (1966).

7. Nilsson, N., "Problem-Solving Methods in Artificial Intelligence," McGraw-Hill (1971).

INITIAL DISTRIBUTION

Copies	
1	DODCI T. Braithwaite
1	ARPA L. Roberts
2	U.S. Army Picatinny Arsenal 1 R. Isakower
1	U.S. Army Frankfort Arsenal D. Frederick
1	USAMERDC J. Marburger
4	CNO 1 OP 916 1 OP 916C1, LCDR Poteat 1 OP 916D 1 OP 098TD, L. Aarons
1	CMC
6	CHONR 1 400R, R. Ryan 1 430, R. Lundegard 1 432, L. Bram 1 437, M. Denicoff 1 437, G. Goldstein
1	DNL
8	CHNAVMAT 1 MAT 0141E, R. Jeske 1 MAT 03 1 MAT 03A, CDR Booth 1 MAT 03L, J. Lawson 1 MAT 03L4, J. Huth 1 MAT 03P2, P. Newton 1 MAT 03P21, S. Atchison
4	USNA 1 D. Rogers 1 A. Adams 1 Dept of Math

Copies	
5	NAVPGSCOL 1 M. Woods 1 D. Williams 1 G. Barksdale 1 C. Comstock
1	NAVWARCOL
1	USNROTC & NAVADMINU, MIT
1	NAVCOSSACT
1	ADPESO
1	CGMCDEC
1	ONR Boston
1	ONR Chicago R. Buchal
1	ONR Pasadena R. Lau
5	NRL 1 5030, S. Wilson 1 5400, B. Wald 1 7810, A. Bligh 1 8050, CDR Tatro
1	COMNAVINT
1	NAVELECSYSCOM
7	NAVSHIPSYSCOM 1 SHIPS 03, RADM Andrews 1 SHIPS 0311, B. Orleans 1 SHIPS 03414, A. Chaikin 1 SHIPS 03423, C. Pohler 1 SHIPS 0719, L. Rosenthal 1 SHIPS 08, Nuclear Power Directorate
3	NAVAIRSYSCOM 1 NAVAIR 5033, R. Saenger 1 NAVAIR 5333F4, R. Entner 1 NAVAIR 5375A, J. Polgren
1	NAVFACENGCOM

Copies	
1	NAVORDSYSCOM
1	NAVAIRDEVCEN
1	CIVENGRLAB
10	NELC 3 5000, A. Beutel 3 5200, M. Lamendola 3 5300, J. Dodds
1	NAVUSEACEN
1	NAVWPNSCEN L. Diesen
1	NAVCOASTSYSLAB
2	NOL 1 H. Stevens
6	NWL 1 Code K 1 Code K-1 1 Code KO 1 Code KP 1 Code KPS
8	NAVSEC 3 SEC 6102C, P. Bono 1 SEC 6114, R. Johnson 1 SEC 6114E, A. Fuller 1 SEC 6178D03, L. Biscomb
1	AFOSR Code 423
1	Rome Air Development Center
1	WPAFB AFFDL
12	DDC
1	NASA Langley Research Center 1 R. Fulton
1	Carnegie-Mellon University Professor A. Newell
1	College of William & Mary Professor N. Gibbs

Copies	
6	MIT 1 Professor M. Minsky 1 Professor T. Winograd 1 Professor P. Winston 1 Dr. A. Nevins 1 D. McDermott 1 G. Sussman
1	MIT Lincoln Laboratory P. Rovner
1	Ohio State University Professor L. White
1	Southern Methodist University Professor R. Korfhage
2	Stanford University 1 Professor J. McCarthy 1 Professor J. Feldman
1	UCLA Professor M. Melkanoff
1	University of Florida Professor J. Tou
1	Hughes Research Laboratory B. Bullock
1	IBM Federal Systems Division J. Sammet
2	IBM Watson Research Laboratory Yorktown Heights, New York 1 G. F. Codd 1 C. H. Thompson
2	Stanford Research Institute 1 Dr. C. Rosen 1 Dr. B. Raphael
1	Systems Development Corporation Santa Monica, California E. Book
1	Xerox Research Laboratories Palo Alto, California 1 Dr. D. Bobrow

CENTER DISTRIBUTION

Copies	Code
1	18/1809
1	1802.1
1	1802.3
1	1802.4
2	1805
2	183
1	1832
1	1833
30	1834
1	1835
2	184
2	185
1	1858
2	186
1	1863
1	1867
2	188
2	189
2	1891, Central Depository

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Naval Ship Research and Development Center Bethesda, Maryland 20034	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

DESIGN TRADE-OFFS FOR A SOFTWARE ASSOCIATIVE MEMORY

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

5. AUTHOR(S) *(First name, middle initial, last name)*

Sidney Berkowitz, Ph.D.

6. REPORT DATE May 1973	7a. TOTAL NO. OF PAGES 56	7b. NO. OF REFS 7

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S) NSRDC Report 3531
b. PROJECT NO. SR0140301	
c. 16565 61151N	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d. 1-1834-001	

10. DISTRIBUTION STATEMENT

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

This report describes an associative (content-addressable) computer memory simulation, called GIRS (Graph Information Retrieval System), designed to handle the dynamic insertion, retrieval, and deletion of arbitrary symbolic or numeric data structures. The main purpose of the study is to demonstrate fundamental trade-offs between time, space, complexity, and flexibility in the field-level operation of any associative memory simulation. Specifically, the paper concludes that

- a reduction of retrieval time is possible at the cost of a complex linkage scheme and slow insertion;
- the design of a random node generator can be optimized to match the scramblin
transformation and reduce retrieval time;
- a dynamic reorganization of pages and the use of inference mechanisms can reduce the number of page fetches and handle complex queries with minimal storage.

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Associative Memory						
Data Base						
Graph						
Data Management						
Hashed Addressing						
Paging						